Readings

This week
- Multi-cycle microarchitecture
  - P&P, Appendices A and C
  - H&H, Chapter 7.4
- Microprogramming
  - P&P, Appendices A and C

Next week
- Pipelining
  - H&H, Chapter 7.5
- Pipelining Issues
  - H&H, Chapter 7.8.1-7.8.3
Agenda for Today & Next Few Lectures

- Instruction Set Architectures (ISA): LC-3 and MIPS
- Assembly programming: LC-3 and MIPS
- Microarchitecture (principles & single-cycle uarch)
- Multi-cycle microarchitecture
- Microprogramming
- Pipelining
- Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, ...
- Out-of-Order Execution
Recall: A Very Basic Instruction Processing Engine

- Each instruction takes a single clock cycle to execute
- Only combinational logic is used to implement instruction execution
  - No intermediate, programmer-invisible state updates

\[
\begin{align*}
\text{AS} &= \text{Architectural (programmer visible) state at the beginning of a clock cycle} \\
\text{Process instruction in one clock cycle} \\
\text{AS}' &= \text{Architectural (programmer visible) state at the end of a clock cycle}
\end{align*}
\]
Recall: Single-Cycle Microarchitecture

- Single-cycle machine

- What is the clock cycle time determined by?
- What is the critical path of the combinational logic determined by?
Recall: Let’s Start with the State Elements

- Data and control inputs

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**
Recall: The Single-Cycle MIPS Datapath

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**

JAL, JR, JALR omitted
Single-Cycle Control Logic
### Single-Cycle Hardwired Control

- As combinational function of $\text{Inst=MEM[PC]}$

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### R-Type

- 6 bits
- 5 bits
- 5 bits
- 5 bits
- 5 bits
- 6 bits
- opcode
- rs
- rt
- rd
- shamt
- funct

#### I-Type

- 6 bits
- 5 bits
- 5 bits
- 16 bits

#### J-Type

- 6 bits
- 26 bits

- Consider
  - All R-type and I-type ALU instructions
  - $\text{lwl}$ and $\text{sw}$
  - $\text{beq}$, $\text{bne}$, $\text{blez}$, $\text{bgtz}$
  - $\text{j}$, $\text{jr}$, $\text{jal}$, $\text{jalr}$
<table>
<thead>
<tr>
<th></th>
<th>When De-asserted</th>
<th>When asserted</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RegDest</strong></td>
<td>GPR write select according to \texttt{rt}, i.e., \texttt{inst[20:16]}</td>
<td>GPR write select according to \texttt{rd}, i.e., \texttt{inst[15:11]}</td>
<td>\texttt{opcode==0}</td>
</tr>
<tr>
<td><strong>ALUSrc</strong></td>
<td>\texttt{2^{nd} ALU input from 2^{nd} GPR read port}</td>
<td>\texttt{2^{nd} ALU input from sign-extended 16-bit immediate}</td>
<td>\texttt{(opcode!=0) &amp;&amp; (opcode!=BEQ) &amp;&amp; (opcode!=BNE)}</td>
</tr>
<tr>
<td><strong>MemtoReg</strong></td>
<td>Steer ALU result to GPR write port</td>
<td>steer memory load to GPR wr. port</td>
<td>\texttt{opcode==LW}</td>
</tr>
<tr>
<td><strong>RegWrite</strong></td>
<td>GPR write disabled</td>
<td>GPR write enabled</td>
<td>\texttt{(opcode!=SW) &amp;&amp; (opcode!=Bxx) &amp;&amp; (opcode!=J) &amp;&amp; (opcode!=JR))</td>
</tr>
</tbody>
</table>

JAL and JALR require additional RegDest and MemtoReg options
## Single-Bit Control Signals (II)

<table>
<thead>
<tr>
<th>PCSrc</th>
<th>When De-asserted</th>
<th>When asserted</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemRead</td>
<td>Memory read disabled</td>
<td>Memory read port return load value</td>
<td>$\text{opcode}==\text{LW}$</td>
</tr>
<tr>
<td>MemWrite</td>
<td>Memory write disabled</td>
<td>Memory write enabled</td>
<td>$\text{opcode}==\text{SW}$</td>
</tr>
<tr>
<td>PCSrc&lt;sub&gt;1&lt;/sub&gt;</td>
<td>According to $\text{PCSsrc}_2$</td>
<td>next PC is based on 26-bit immediate jump target</td>
<td>$(\text{opcode}==\text{J}) \text{</td>
</tr>
<tr>
<td>PCSrc&lt;sub&gt;2&lt;/sub&gt;</td>
<td>next PC = PC + 4</td>
<td>next PC is based on 16-bit immediate branch target</td>
<td>$(\text{opcode}==\text{Bxx}) \text{ &amp;&amp; } \text{“bcond is satisfied”}$</td>
</tr>
</tbody>
</table>

JR and JALR require additional PCSrc options
ALU Control

- case *opcode*
  - ‘0’ ⇒ select operation according to *funct*
  - ‘ALUi’ ⇒ selection operation according to *opcode*
  - ‘LW’ ⇒ select addition
  - ‘SW’ ⇒ select addition
  - ‘Bxx’ ⇒ select bcond generation function
  - ____ ⇒ don’t care

- Example ALU operations
  - ADD, SUB, AND, OR, XOR, NOR, etc.
  - bcond on equal, not equal, LE zero, GT zero, etc.
Let’s Control The Single-Cycle MIPS Datapath

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

JAL, JR, JALR omitted
R-Type ALU

- Instruction [31–0]
  - PC+4 [31–28]
  - Instruction [31–26]
  - Instruction [25–0]
    - Shift left 2
    - Jump address [31–0]
  - Instruction [25–21]
  - Instruction [20–16]
  - Instruction [15–11]
  - Instruction [15–0]
  - Instruction [5–0]
- Add
- ALU operation
- Control
- RegDst
- Jump
- Branch
- MemRead
- MemtoReg
- ALUOp
- MemWrite
- ALUSrc
- RegWrite
- ALU

- PCSrc₁ = Jump
- PCSrc₂ = Br Taken

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**
I-Type ALU

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**
Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
Branch (Not Taken)

Some control signals are dependent on the processing of data

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**
Some control signals are dependent on the processing of data.
What is in That Control Box?

- **Combinational Logic** → **Hardwired Control**
  - Idea: Control signals generated combinationally based on instruction
  - Necessary in a single-cycle microarchitecture

- **Sequential Logic** → **Sequential/Microprogrammed Control**
  - Idea: A memory structure contains the control signals associated with an instruction
  - Control Store
Review: Complete Single-Cycle Processor

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

JAL, JR, JALR omitted
Another Single-Cycle MIPS Processor (from H&H)

See backup slides to reinforce the concepts we have covered. They are to complement your reading:

H&H, Chapter 7.1-7.3, 7.6
Another Complete Single-Cycle Processor

Single-cycle processor. Harris and Harris, Chapter 7.3.
Example: Single-Cycle Datapath: `lw` fetch

**STEP 1:** Fetch instruction

```
lw $s3, 1($0)  # read memory word 1 into $s3
```

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Single-Cycle Datapath: lw register read

**STEP 2:** Read source operands from register file

lw $s3, 1($0)  # read memory word 1 into $s3

**I-Type**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Single-Cycle Datapath: lw immediate

**STEP 3:** Sign-extend the immediate

```
lw $s3, 1($0)    # read memory word 1 into $s3
```

**I-Type**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
**Single-Cycle Datapath: lw address**

- **STEP 4:** Compute the memory address

```
lw $s3, 1($0)  # read memory word 1 into $s3
```

**I-Type**

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Single-Cycle Datapath: lw memory read

- **STEP 5:** Read from memory and write back to register file

lw $s3, 1($0)  # read memory word 1 into $s3

**I-Type**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Single-Cycle Datapath: lw PC increment

- **STEP 6:** Determine address of next instruction

```
lw $s3, 1($0)  # read memory word 1 into $s3
```

**I-Type**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>imm</td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Similarly, We Need to Design the Control Unit

- **Control signals** generated by the decoder in control unit

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op5:0</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>AluSrc</th>
<th>Branch</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp1:0</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>000000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>100011</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>101011</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>000100</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>addi</td>
<td>001000</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>j</td>
<td>000010</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>XX</td>
<td>1</td>
</tr>
</tbody>
</table>

Single-cycle processor. Harris and Harris, Chapter 7.3.
Another Complete Single-Cycle Processor (H&H)
Your Assignment

- Please read the Backup Slides

- Please do your readings from the H&H Book
  - H&H, Chapter 7.1-7.3, 7.6
Single-Cycle Uarch I (We Developed in Lectures)

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

JAL, JR, JALR omitted
Single-Cycle Uarch II (In Your Readings)
Evaluating the Single-Cycle Microarchitecture
A Single-Cycle Microarchitecture

- Is this a good idea/design?
- When is this a good design?
- When is this a bad design?
- How can we design a better microarchitecture?
Performance Analysis Basics
Processor Performance

How fast is my program?
- Every program consists of a series of instructions
- Each instruction needs to be executed.
Processor Performance

- How fast is my program?
  - Every program consists of a series of instructions
  - Each instruction needs to be executed.

- So how fast are my instructions?
  - Instructions are realized on the hardware
  - They can take one or more clock cycles to complete
  - *Cycles per Instruction = CPI*
 Processor Performance

■ How fast is my program?
  ▪ Every program consists of a series of instructions
  ▪ Each instruction needs to be executed.

■ So how fast are my instructions?
  ▪ Instructions are realized on the hardware
  ▪ They can take one or more clock cycles to complete
  ▪ $Cycles\ per\ Instruction = CPI$

■ How much time is one clock cycle?
  ▪ The critical path determines how much time one cycle requires = \textit{clock period}.
  ▪ $1/clock\ period = clock\ frequency$ = how many cycles can be done each second.
Processor Performance

- **Now as a general formula**
  - Our program consists of executing \( N \) instructions.
  - Our processor needs \( CPI \) cycles for each instruction.
  - The maximum clock speed of the processor is \( f \), and the clock period is therefore \( T=1/f \)
Processor Performance

- **Now as a general formula**
  - Our program consists of executing \( N \) instructions.
  - Our processor needs \( \text{CPI} \) cycles for each instruction.
  - The maximum clock speed of the processor is \( f \), and the clock period is therefore \( T = 1/f \)

- **Our program executes in**

\[
N \times \text{CPI} \times (1/f) = \\
N \times \text{CPI} \times T \text{ seconds}
\]
**Performance Analysis Basics**

- Execution time of an instruction
  - \( \{\text{CPI}\} \times \{\text{clock cycle time}\} \)
  - CPI: Number of cycles it takes to execute an instruction

- Execution time of a program
  - Sum over all instructions \([\{\text{CPI}\} \times \{\text{clock cycle time}\}]\)
  - \( \{\text{# of instructions}\} \times \{\text{Average CPI}\} \times \{\text{clock cycle time}\} \)
Performance Analysis of Our Single-Cycle Design
A Single-Cycle Microarchitecture: Analysis

- Every instruction takes 1 cycle to execute
  - CPI (Cycles per instruction) is strictly 1

- How long each instruction takes is determined by how long the slowest instruction takes to execute
  - Even though many instructions do not need that long to execute

- Clock cycle time of the microarchitecture is determined by how long it takes to complete the slowest instruction
  - Critical path of the design is determined by the processing time of the slowest instruction
What is the Slowest Instruction to Process?

- Let’s go back to the basics

- All six phases of the instruction processing cycle take a single machine clock cycle to complete
  
  - **Fetch**
  - **Decode**
  - **Evaluate Address**
  - **Fetch Operands**
  - **Execute**
  - **Store Result**

- Do each of the above phases take the same time (latency) for all instructions?
Let’s Find the Critical Path

[Diagram with labels and connections]
### Example Single-Cycle Datapath Analysis

- Assume (for the design in the previous slide)
  - memory units (read or write): 200 ps
  - ALU and adders: 100 ps
  - register file (read or write): 50 ps
  - other combinational logic: 0 ps

<table>
<thead>
<tr>
<th>steps</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>resources</td>
<td>mem</td>
<td>RF</td>
<td>ALU</td>
<td>mem</td>
<td>RF</td>
<td></td>
</tr>
<tr>
<td>R-type</td>
<td>200</td>
<td>50</td>
<td>100</td>
<td>50</td>
<td>400</td>
<td></td>
</tr>
<tr>
<td>I-type</td>
<td>200</td>
<td>50</td>
<td>100</td>
<td>50</td>
<td>400</td>
<td></td>
</tr>
<tr>
<td>LW</td>
<td>200</td>
<td>50</td>
<td>100</td>
<td>200</td>
<td>600</td>
<td></td>
</tr>
<tr>
<td>SW</td>
<td>200</td>
<td>50</td>
<td>100</td>
<td>200</td>
<td>550</td>
<td></td>
</tr>
<tr>
<td>Branch</td>
<td>200</td>
<td>50</td>
<td>100</td>
<td></td>
<td>350</td>
<td></td>
</tr>
<tr>
<td>Jump</td>
<td>200</td>
<td></td>
<td></td>
<td></td>
<td>200</td>
<td></td>
</tr>
</tbody>
</table>
Let’s Find the Critical Path

[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
R-Type and I-Type ALU

[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
Branch Taken

[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
Jump

[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
What About Control Logic?

- How does that affect the critical path?

- Food for thought for you:
  - Can control logic be on the critical path?
  - Historical example:
    - CDC 5600: control store access too long...
What is the Slowest Instruction to Process?

- Memory is not magic
- What if memory *sometimes* takes 100ms to access?
- Does it make sense to have a simple register to register add or jump to take \{100ms+all else to do a memory operation\}?
- And, what if you need to access memory more than once to process an instruction?
  - Which instructions need this?
  - Do you provide multiple ports to memory?
Single Cycle uArch: Complexity

- Contrived
  - All instructions run as slow as the slowest instruction

- Inefficient
  - All instructions run as slow as the slowest instruction
  - Must provide worst-case combinational resources in parallel as required by any instruction
  - Need to replicate a resource if it is needed more than once by an instruction during different parts of the instruction processing cycle

- Not necessarily the simplest way to implement an ISA
  - Single-cycle implementation of REP MOVS (x86) or INDEX (VAX)?

- Not easy to optimize/improve performance
  - Optimizing the common case does not work (e.g. common instructions)
  - Need to optimize the worst case all the time
(Micro)architecture Design Principles

- Critical path design
  - Find and decrease the maximum combinational logic delay
  - Break a path into multiple cycles if it takes too long

- Bread and butter (common case) design
  - Spend time and resources on where it matters most
    - i.e., improve what the machine is really designed to do
  - Common case vs. uncommon case

- Balanced design
  - Balance instruction/data flow through hardware components
  - Design to eliminate bottlenecks: balance the hardware for the work
Single-Cycle Design vs. Design Principles

- Critical path design
- Bread and butter (common case) design
- Balanced design

How does a single-cycle microarchitecture fare in light of these principles?
Aside: System Design Principles

- When designing computer systems/architectures, it is important to follow good principles.

- Remember: “principled design” from our first lecture
  - Frank Lloyd Wright: “architecture [...] based upon principle, and not upon precedent”
Aside: From Lecture 1

“architecture [...] based upon principle, and not upon precedent”
Aside: System Design Principles

- We will continue to cover key principles in this course
- Here are some references where you can learn more

- Gene M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS Conference, April 1967. (Amdahl’s Law→Common-case design)
A Key System Design Principle

- Keep it simple

- “Everything should be made as simple as possible, but no simpler.”
  - Albert Einstein

- And, keep it low cost: “An engineer is a person who can do for a dime what any fool can do for a dollar.”

- For more, see:
Multi-Cycle Microarchitectures
Multi-Cycle Microarchitectures

- Goal: Let each instruction take (close to) only as much time it really needs

- Idea
  - Determine clock cycle time independently of instruction processing time
  - Each instruction takes as many clock cycles as it needs to take
    - Multiple state transitions per instruction
    - The states followed by each instruction is different
Remember: The “Process instruction” Step

- ISA specifies abstractly what AS’ should be, given an instruction and AS
  - It defines an abstract finite state machine where
    - State = programmer-visible state
    - Next-state logic = instruction execution specification
  - From ISA point of view, there are no “intermediate states” between AS and AS’ during instruction execution
    - One state transition per instruction

- Microarchitecture implements how AS is transformed to AS’
  - There are many choices in implementation
  - We can have programmer-invisible state to optimize the speed of instruction execution: multiple state transitions per instruction
    - Choice 1: AS $\rightarrow$ AS’ (transform AS to AS’ in a single clock cycle)
    - Choice 2: AS $\rightarrow$ AS+MS1 $\rightarrow$ AS+MS2 $\rightarrow$ AS+MS3 $\rightarrow$ AS’ (take multiple clock cycles to transform AS to AS’)

67
Multi-Cycle Microarchitecture

AS = Architectural (programmer visible) state at the beginning of an instruction

Step 1: Process part of instruction in one clock cycle

Step 2: Process part of instruction in the next clock cycle

... 

AS’ = Architectural (programmer visible) state at the end of a clock cycle
Benefits of Multi-Cycle Design

- **Critical path design**
  - Can keep reducing the critical path independently of the worst-case processing time of any instruction

- **Bread and butter (common case) design**
  - Can optimize the number of states it takes to execute “important” instructions that make up much of the execution time

- **Balanced design**
  - No need to provide more capability or resources than really needed
    - An instruction that needs resource X multiple times does not require multiple X’s to be implemented
    - Leads to more efficient hardware: Can reuse hardware components needed multiple times for an instruction
Downsides of Multi-Cycle Design

- Need to store the intermediate results at the end of each clock cycle
  - Hardware overhead for registers
  - Register setup/hold overhead paid multiple times for an instruction
Remember: Performance Analysis

- Execution time of an instruction
  - \(\{\text{CPI}\} \times \{\text{clock cycle time}\}\)

- Execution time of a program
  - Sum over all instructions \([\{\text{CPI}\} \times \{\text{clock cycle time}\}]\)
  - \(\{\text{# of instructions}\} \times \{\text{Average CPI}\} \times \{\text{clock cycle time}\}\)

- Single cycle microarchitecture performance
  - CPI = 1
  - Clock cycle time = long

- Multi-cycle microarchitecture performance
  - CPI = different for each instruction
    - Average CPI → hopefully small
  - Clock cycle time = short

We have two degrees of freedom to optimize independently
Not easy to optimize design
A Multi-Cycle Microarchitecture

A Closer Look
How Do We Implement This?


An elegant implementation:
- The concept of microcoded/microprogrammed machines
Key Idea for Realization

- One can implement the “process instruction” step as a finite state machine that sequences between states and eventually returns back to the “fetch instruction” state.

- A state is defined by the control signals asserted in it.

- Control signals for the next state are determined in current state.
The Instruction Processing Cycle

- Fetch
- Decode
- Evaluate Address
- Fetch Operands
- Execute
- Store Result
A Basic Multi-Cycle Microarchitecture

- Instruction processing cycle divided into “states”
  - A stage in the instruction processing cycle can take multiple states

- A multi-cycle microarchitecture sequences from state to state to process an instruction
  - The behavior of the machine in a state is completely determined by control signals in that state

- The behavior of the entire processor is specified fully by a finite state machine

- In a state (clock cycle), control signals control two things:
  - How the datapath should process the data
  - How to generate the control signals for the (next) clock cycle
One Example Multi-Cycle Microarchitecture
Remember: Single-Cycle MIPS Processor
Multi-cycle MIPS Processor

- **Single-cycle microarchitecture:**
  - simple
  - cycle time limited by longest instruction (lw)
  - three adders/ALUs and two memories

- **Multi-cycle microarchitecture:**
  - higher clock speed
  - simpler instructions run faster
  - reuse expensive hardware on multiple cycles
  - sequencing overhead paid many times
  - hardware overhead for storing intermediate results

- **Same design steps: datapath & control**
What Do We Want To Optimize

- Single Cycle Architecture uses two memories
  - One memory stores instructions, the other data
  - We want to use a single memory (Smaller size)
What Do We Want To Optimize

- Single Cycle Architecture uses two memories
  - One memory stores instructions, the other data
  - We want to use a single memory (Smaller size)

- Single Cycle Architecture needs three adders
  - ALU, PC, Branch address calculation
  - We want to use the ALU for all operations (smaller size)
What Do We Want To Optimize

- **Single Cycle Architecture uses two memories**
  - One memory stores instructions, the other data
  - We want to use a single memory (Smaller size)

- **Single Cycle Architecture needs three adders**
  - ALU, PC, Branch address calculation
  - We want to use the ALU for all operations (smaller size)

- **In Single Cycle Architecture all instructions take one cycle**
  - The most complex operation slows down everything!
  - Divide all instructions into multiple steps
  - Simpler instructions can take fewer cycles (average case may be faster)
Consider the \texttt{lw} instruction

\begin{itemize}
\item For an instruction such as: \texttt{lw \ $t0, \ 0x20($t1)}
\end{itemize}

\begin{itemize}
\item We need to:
\begin{itemize}
\item Read the instruction from memory
\item Then read $t1$ from register array
\item Add the immediate value ($0x20$) to calculate the memory address
\item Read the content of this address
\item Write to the register $t0$ this content
\end{itemize}
\end{itemize}
Multi-cycle Datapath: instruction fetch

- First consider executing lw
  - STEP 1: Fetch instruction

read from the memory location \([rs]+imm\) to location \([rt]\)

**I-Type**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Multi-cycle Datapath: lw register read

**I-Type**

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
<td></td>
</tr>
</tbody>
</table>
Multi-cycle Datapath: lw immediate

I-Type

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Multi-cycle Datapath: lw address

I-Type

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Multi-cycle Datapath: lw memory read

I-Type

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Multi-cycle Datapath: lw write register

I-Type

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Multi-cycle Datapath: increment PC
Multi-cycle Datapath: Sw

- Write data in rt to memory
Multi-cycle Datapath: R-type Instructions

- Read from rs and rt
  - Write ALUResult to register file
  - Write to rd (instead of rt)
Multi-cycle Datapath: beq

- Determine whether values in rs and rt are equal
  - Calculate branch target address:
    
    \[ BTA = (\text{sign-extended immediate } \ll 2) + (\text{PC}+4) \]
Complete Multi-cycle Processor
Control Unit

Control Unit

Main Controller (FSM)

Opcode_{5:0}

ALUOp_{1:0}

Funct_{5:0}

Decoder

RegWrite

IRWrite

MemWrite

PCWrite

Branch

MemtoReg

RegDst

IorD

ALUSrcB_{1:0}

ALUSrcA

ALU Control_{2:0}

Register Enables

Multiplexer Selects
Main Controller FSM: Fetch
Main Controller FSM: Fetch

S0: Fetch
Reset
IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 0
IRWrite
PCWrite

IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 0
IRWrite
PCWrite
Main Controller FSM: Decode

S0: Fetch
- IorD = 0
- AluSrcA = 0
- ALUSrcB = 01
- ALUOp = 00
- PCSrc = 0
- IRWrite
- PCWrite

S1: Decode

Reset

Diagram of the controller FSM with states and signals labeled.
Main Controller FSM: Address Calculation

IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 0
IRWrite
PCWrite

S0: Fetch
S1: Decode
S2: MemAdr

Op = LW
or
Op = SW

Reset
Main Controller FSM: Address Calculation

S0: Fetch
IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
IRWrite
PCSrc = 0
IRWrite
PCWrite

S1: Decode
Op = LW
or
Op = SW

S2: MemAdr
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
Main Controller FSM: \texttt{lw}

\begin{itemize}
\item \texttt{Reset}
\item \texttt{S0: Fetch}
\item \texttt{S1: Decode}
\item \texttt{S2: MemAdr}
\item \texttt{S3: MemRead}
\item \texttt{S4: MemWriteback}
\end{itemize}

- \texttt{IorD} = 0
- \texttt{AluSrcA} = 0
- \texttt{ALUSrcB} = 01
- \texttt{ALUOp} = 00
- \texttt{PCSrc} = 0
- \texttt{IRWrite}
- \texttt{PCWrite}

- \texttt{IorD} = 1
- \texttt{RegDst} = 0
- \texttt{MemtoReg} = 1
- \texttt{RegWrite}

- \texttt{Op} = \texttt{LW}
- \texttt{Op} = \texttt{SW}

- \texttt{Op} = \texttt{LW} or \texttt{SW}
Main Controller FSM: sw

S0: Fetch
- Reset
- IorD = 0
- AluSrcA = 0
- ALUSrcB = 01
- ALUOp = 00
- PCSrc = 0
- IRWrite
- PCWrite

S1: Decode
- Op = LW
- or
- Op = SW

S2: MemAdr
- ALUSrcA = 1
- ALUSrcB = 10
- ALUOp = 00

S3: MemRead
- Op = LW
- IorD = 1

S4: MemWriteback
- RegDst = 0
- MemtoReg = 1
- MemWrite

S5: MemWrite
- IorD = 1
- MemWrite
Main Controller FSM: R-Type

S0: Fetch

Reset

S1: Decode

IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 0
IRWrite
PCWrite

Op = LW
or
Op = SW

S2: MemAdr

ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00

S3: MemRead

Op = LW

S4: MemWriteback

IorD = 1

S5: MemWrite

Op = SW

S6: Execute

ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10

S7: ALU Writeback

RegDst = 1
MemtoReg = 0
RegWrite

S4: Mem Writeback

IorD = 1
MemWrite

RegDst = 0
MemtoReg = 1
RegWrite

Op = R-type

LW
op

SW
Main Controller FSM: beq
Complete Multi-cycle Controller FSM

S0: Fetch

IorD = 0
AluSrcA = 0
AlUSrcB = 01
aluOp = 00
IRWrite
PCWrite

S1: Decode

ALUSrcA = 0
ALUSrcB = 0
aluOp = 00

S2: MemAdr

ALUSrcA = 1
ALUSrcB = 00
aluOp = 00
Op = LW
or
Op = SW

S3: MemRead

IorD = 1

S4: MemWriteback

RegDst = 0
MemtoReg = 1
RegWrite

S5: MemWrite

IorD = 1
MemWrite

S6: Execute

RegDst = 1
MemtoReg = 0
RegWrite

S7: ALU Writeback

RegDst = 1
MemtoReg = 0
RegWrite

S8: Branch

ALUSrcA = 1
ALUSrcB = 00
aluOp = 01
PCSrc = 1
Branch

Op = BEQ

S0: Fetch

IorD = 0
AluSrcA = 0
AlUSrcB = 01
aluOp = 00
IRWrite
PCWrite

S1: Decode

ALUSrcA = 0
ALUSrcB = 0
aluOp = 00

S2: MemAdr

ALUSrcA = 1
ALUSrcB = 00
aluOp = 00
Op = LW
or
Op = SW

S3: MemRead

IorD = 1

S4: MemWriteback

RegDst = 0
MemtoReg = 1
RegWrite

S5: MemWrite

IorD = 1
MemWrite

S6: Execute

RegDst = 1
MemtoReg = 0
RegWrite

S7: ALU Writeback

RegDst = 1
MemtoReg = 0
RegWrite

S8: Branch

ALUSrcA = 1
ALUSrcB = 00
aluOp = 01
PCSrc = 1
Branch

Op = BEQ
Main Controller FSM: addi

S0: Fetch
- \text{Reset}
- \text{Op = LW or SW}

S1: Decode
- \text{Op = R-type}

S2: MemAdr
- \text{ALUSrcA = 1, ALUSrcB = 10, ALUOp = 00}
  - \text{Op = LW}

S3: MemRead
- \text{IorD = 1}

S4: MemWriteback
- \text{RegDst = 0, MemtoReg = 1, RegWrite}

S5: MemWrite
- \text{IorD = 1, MemWrite}

S6: Execute
- \text{ALUSrcA = 0, ALUSrcB = 11, ALUOp = 00}
  - \text{Op = BEQ}

S7: ALU Writeback
- \text{RegDst = 1, MemtoReg = 0, RegWrite}

S8: Branch
- \text{ALUSrcA = 1, ALUSrcB = 00, ALUOp = 01}
- \text{PCSrc = 1, Branch}

S9: addi Execute
- \text{Op = ADDI}

S10: addi Writeback

Main Controller FSM: addi

S0: Fetch
- IorD = 0
- AluSrcA = 0
- AluSrcB = 01
- AluOp = 00
- PCSrc = 0
- IRWrite
- PCWrite

S1: Decode
- ALUSrcA = 0
- ALUSrcB = 01
- ALUOp = 00
- IRWrite
- PCWrite

S2: MemAdr
- ALUSrcA = 1
- ALUSrcB = 10
- ALUOp = 00

S3: MemRead
- IorD = 1

S4: MemWriteback
- RegDst = 0
- MemtoReg = 1
- RegWrite

S5: MemWrite
- IorD = 1

S6: Execute
- RegDst = 0
- MemtoReg = 0
- RegWrite

S7: ALU Writeback
- Op = ADDI

S8: Branch
- Op = BEQ

S9: ADDI Execute
- RegDst = 0
- MemtoReg = 0
- RegWrite

S10: ADDI Writeback
Extended Functionality: j
Control FSM: \( j \)
Control FSM: \( j \)
Backup Slides on Single-Cycle Uarch for Your Own Study

Please study these to reinforce the concepts we covered in lectures.

Please do the readings together with these slides:
H&H, Chapter 7.1-7.3, 7.6
Another Single-Cycle MIPS Processor (from H&H)

These are slides for your own study.
They are to complement your reading
H&H, Chapter 7.1-7.3, 7.6
What to do with the Program Counter?

- The PC needs to be incremented by 4 during each cycle (for the time being).
- Initial PC value (after reset) is 0x00400000

```verilog
reg [31:0] PC_p, PC_n;  // Present and next state of PC

// [...]

assign PC_n <= PC_p + 4;  // Increment by 4;

always @ (posedge clk, negedge rst)
begin
  if (rst == '0') PC_p <= 32'h00400000;  // default
  else          PC_p <= PC_n;            // when clk
end
```
We Need a Register File

- **Store 32 registers, each 32-bit**
  - \(2^5 = 32\), we need 5 bits to address each

- **Every R-type instruction uses 3 register**
  - Two for reading (RS, RT)
  - One for writing (RD)

- **We need a special memory with:**
  - 2 read ports (address x2, data out x2)
  - 1 write port (address, data in)
Register File

input [4:0]  a_rs, a_rt, a_rd;
input [31:0]  di_rd;
input        we_rd;
output [31:0] do_rs, do_rt;

reg [31:0] R_arr [31:0]; // Array that stores regs

// Circuit description
assign do_rs = R_arr[a_rs];  // Read RS

assign do_rt = R_arr[a_rt];  // Read RT

always @ (posedge clk)
if (we_rd) R_arr[a_rd] <= di_rd; // write RD
Register File

```verilog
input [4:0] a_rs, a_rt, a_rd;
input [31:0] di_rd;
input    we_rd;
output [31:0] do_rs, do_rt;

reg [31:0] R_arr [31:0]; // Array that stores regs

// Circuit description; add the trick with $0
assign do_rs = (a_rs != 5'b00000)? // is address 0?
    R_arr[a_rs] : 0; // Read RS or 0
assign do_rt = (a_rt != 5'b00000)? // is address 0?
    R_arr[a_rt] : 0; // Read RT or 0

always @ (posedge clk)
    if (we_rd) R_arr[a_rd] <= di_rd; // write RD
```
Data Memory Example

- Will be used to store the bulk of data

```verilog
input [15:0] addr; // Only 16 bits in this example
input [31:0] di;
input we;
output [31:0] do;

reg [65535:0] M_arr [31:0]; // Array for Memory

// Circuit description
assign do = M_arr[addr]; // Read memory

always @(posedge clk)
  if (we) M_arr[addr] <= di; // write memory
```
Single-Cycle Datapath: lw fetch

**STEP 1:** Fetch instruction

```
lw $s3, 1($0) # read memory word 1 into $s3
```

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>len</td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
**Single-Cycle Datapath: lw register read**

- **STEP 2:** Read source operands from register file

```
lw $s3, 1($0)  # read memory word 1 into $s3
```

### I-Type

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>imm</td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Single-Cycle Datapath: \texttt{lw} immediate

- \textbf{STEP 3:} Sign-extend the immediate

\begin{center}
\begin{tabular}{|c|c|c|c|}
\hline
\textbf{op} & \textbf{rs} & \textbf{rt} & \textbf{imm} \\
6 bits & 5 bits & 5 bits & 16 bits \\
\hline
\end{tabular}
\end{center}

\texttt{lw} $\$s3, 1($0) \# \text{read memory word 1 into $\$s3$}
Single-Cycle Datapath: \texttt{lw} address

\textbf{STEP 4:} Compute the memory address

\texttt{lw} $\texttt{s3}, 1(\texttt{0})$ \# read memory word 1 into $\texttt{s3}$

\textbf{I-Type}

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Single-Cycle Datapath: lw memory read

**STEP 5:** Read from memory and write back to register file

 lw $s3, 1($0)  # read memory word 1 into $s3

**I-Type**

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Single-Cycle Datapath: \texttt{lw} PC increment

\textbf{STEP 6:} Determine address of next instruction

\begin{align*}
\texttt{lw} \hspace{1em} & \hspace{1em} \texttt{$s3, 1($0)} & \# \text{ read memory word 1 into } \texttt{$s3} \\
\end{align*}

\textbf{I-Type}

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Single-Cycle Datapath: `sw`

- **Write data in `rt` to memory**

```plaintext
sw $t7, 44($0)  # write t7 into memory address 44
```

**I-Type**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Single-Cycle Datapath: R-type Instructions

- Read from rs and rt, write ALUResult to register file

\[ \text{add } t, b, c \quad \text{# } t = b + c \]

**R-Type**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>
Single-Cycle Datapath: beq

- Determine whether values in rs and rt are equal
- Calculate $BTA = (\text{sign-extended immediate} \ll 2) + (\text{PC}+4)$
Complete Single-Cycle Processor
Our MIPS Datapath has Several Options

- **ALU inputs**
  - Either RT or Immediate (*MUX*)

- **Write Address of Register File**
  - Either RD or RT (*MUX*)

- **Write Data In of Register File**
  - Either ALU out or Data Memory Out (*MUX*)

- **Write enable of Register File**
  - Not always a register write (*MUX*)

- **Write enable of Memory**
  - Only when writing to memory (sw) (*MUX*)

*All these options are our control signals*
Control Unit

- Opcode
- MemtoReg
- MemWrite
- Branch
- ALUSrc
- RegDst
- RegWrite

Main Decoder

ALU Decoder

ALUOp  | Meaning
--- | ---
00 | add
01 | subtract
10 | look at funct field
11 | n/a
ALU Does the Real Work in a Processor

<table>
<thead>
<tr>
<th>$F_{2:0}$</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>A &amp; B</td>
</tr>
<tr>
<td>001</td>
<td>A</td>
</tr>
<tr>
<td>010</td>
<td>A + B</td>
</tr>
<tr>
<td>011</td>
<td>not used</td>
</tr>
<tr>
<td>100</td>
<td>A &amp; ~B</td>
</tr>
<tr>
<td>101</td>
<td>A</td>
</tr>
<tr>
<td>110</td>
<td>A - B</td>
</tr>
<tr>
<td>111</td>
<td>SLT</td>
</tr>
</tbody>
</table>
ALU Internals

<table>
<thead>
<tr>
<th>$F_{2:0}$</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>$A &amp; B$</td>
</tr>
<tr>
<td>001</td>
<td>$A \mid B$</td>
</tr>
<tr>
<td>010</td>
<td>$A + B$</td>
</tr>
<tr>
<td>011</td>
<td>not used</td>
</tr>
<tr>
<td>100</td>
<td>$A &amp; \neg B$</td>
</tr>
<tr>
<td>101</td>
<td>$A \mid \neg B$</td>
</tr>
<tr>
<td>110</td>
<td>$A - B$</td>
</tr>
<tr>
<td>111</td>
<td>SLT</td>
</tr>
</tbody>
</table>
Control Unit: ALU Decoder

<table>
<thead>
<tr>
<th>ALUOp&lt;sub&gt;1:0&lt;/sub&gt;</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Add</td>
</tr>
<tr>
<td>01</td>
<td>Subtract</td>
</tr>
<tr>
<td>10</td>
<td>Look at Funct</td>
</tr>
<tr>
<td>11</td>
<td>Not Used</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALUOp&lt;sub&gt;1:0&lt;/sub&gt;</th>
<th>Funct</th>
<th>ALUControl&lt;sub&gt;2:0&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>X</td>
<td>010 (Add)</td>
</tr>
<tr>
<td>X1</td>
<td>X</td>
<td>110 (Subtract)</td>
</tr>
<tr>
<td>1X</td>
<td>100000 (add)</td>
<td>010 (Add)</td>
</tr>
<tr>
<td>1X</td>
<td>100010 (sub)</td>
<td>110 (Subtract)</td>
</tr>
<tr>
<td>1X</td>
<td>100100 (and)</td>
<td>000 (And)</td>
</tr>
<tr>
<td>1X</td>
<td>100101 (or)</td>
<td>001 (Or)</td>
</tr>
<tr>
<td>1X</td>
<td>101010 (slt)</td>
<td>111 (SLT)</td>
</tr>
</tbody>
</table>
Let us Develop our Control Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op_{5:0}</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>AluSrc</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **RegWrite**: Write enable for the register file
- **RegDst**: Write to register RD or RT
- **AluSrc**: ALU input RT or immediate
- **MemWrite**: Write Enable
- **MemtoReg**: Register data in from Memory or ALU
- **ALUOp**: What operation does ALU do
Let us Develop our Control Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op&lt;sub&gt;5:0&lt;/sub&gt;</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>AluSrc</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>000000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>funct</td>
</tr>
</tbody>
</table>

- **RegWrite**: Write enable for the register file
- **RegDst**: Write to register RD or RT
- **AluSrc**: ALU input RT or immediate
- **MemWrite**: Write Enable
- **MemtoReg**: Register data in from Memory or ALU
- **ALUOp**: What operation does ALU do
Let us Develop our Control Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op$_{5:0}$</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>AluSrc</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>000000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>funct</td>
</tr>
<tr>
<td>lw</td>
<td>100011</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>add</td>
</tr>
</tbody>
</table>

- **RegWrite**: Write enable for the register file
- **RegDst**: Write to register RD or RT
- **AluSrc**: ALU input RT or immediate
- **MemWrite**: Write Enable
- **MemtoReg**: Register data in from Memory or ALU
- **ALUOp**: What operation does ALU do
Let us Develop our Control Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op&lt;sub&gt;5:0&lt;/sub&gt;</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>AluSrc</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>000000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>funct</td>
</tr>
<tr>
<td>lw</td>
<td>100011</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>add</td>
</tr>
<tr>
<td>sw</td>
<td>101011</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>add</td>
</tr>
</tbody>
</table>

- **RegWrite**: Write enable for the register file
- **RegDst**: Write to register RD or RT
- **AluSrc**: ALU input RT or immediate
- **MemWrite**: Write Enable
- **MemtoReg**: Register data in from Memory or ALU
- **ALUOp**: What operation does ALU do
## More Control Signals

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op_{5:0}</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>AluSrc</th>
<th>Branch</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>000000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>100011</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>sw</td>
<td>101011</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>add</td>
</tr>
<tr>
<td>beq</td>
<td>000100</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>sub</td>
</tr>
</tbody>
</table>

### New Control Signal

- **Branch**: Are we jumping or not?
### Control Unit: Main Decoder

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op_{5:0}</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>AluSrc</th>
<th>Branch</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp_{1:0}</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>000000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>lw</td>
<td>100011</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td>sw</td>
<td>101011</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>00</td>
</tr>
<tr>
<td>beq</td>
<td>000100</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>01</td>
</tr>
</tbody>
</table>
Single-Cycle Datapath Example: or
Extended Functionality: addi

- No change to datapath
## Control Unit: addi

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op&lt;sub&gt;5:0&lt;/sub&gt;</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>AluSrc</th>
<th>Branch</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp&lt;sub&gt;1:0&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>000000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>lw</td>
<td>100011</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td>sw</td>
<td>101011</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>00</td>
</tr>
<tr>
<td>beq</td>
<td>000100</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>01</td>
</tr>
<tr>
<td>addi</td>
<td>001000</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
</tbody>
</table>
Extended Functionality: $j$
# Control Unit: Main Decoder

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op_{5:0}</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>AluSrc</th>
<th>Branch</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp_{1:0}</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>000000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>100011</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>101011</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>000100</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>j</td>
<td>000100</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>XX</td>
<td>1</td>
</tr>
</tbody>
</table>
Review: Complete Single-Cycle Processor (H&H)
A Bit More on
Performance Analysis
Processor Performance

- How fast is my program?
  - Every program consists of a series of instructions
  - Each instruction needs to be executed.
Processor Performance

How fast is my program?
- Every program consists of a series of instructions
- Each instruction needs to be executed.

So how fast are my instructions?
- Instructions are realized on the hardware
- They can take one or more clock cycles to complete
- *Cycles per Instruction = CPI*
Processor Performance

- **How fast is my program?**
  - Every program consists of a series of instructions
  - Each instruction needs to be executed.

- **So how fast are my instructions?**
  - Instructions are realized on the hardware
  - They can take one or more clock cycles to complete
  - *Cycles per Instruction = CPI*

- **How much time is one clock cycle?**
  - The critical path determines how much time one cycle requires = *clock period*.
  - $1/$clock period = *clock frequency* = how many cycles can be done each second.
Performance Analysis

- Execution time of an instruction
  - \( \{\text{CPI}\} \times \{\text{clock cycle time}\} \)

- Execution time of a program
  - Sum over all instructions \( \{\text{CPI}\} \times \{\text{clock cycle time}\} \)
  - \( \{\text{# of instructions}\} \times \{\text{Average CPI}\} \times \{\text{clock cycle time}\} \)
Processor Performance

Now as a general formula

- Our program consists of executing $N$ instructions.
- Our processor needs $CPI$ cycles for each instruction.
- The maximum clock speed of the processor is $f$, and the clock period is therefore $T=1/f$
Processor Performance

■ Now as a general formula
  ▪ Our program consists of executing \( N \) instructions.
  ▪ Our processor needs \( \text{CPI} \) cycles for each instruction.
  ▪ The maximum clock speed of the processor is \( f \), and the clock period is therefore \( T=1/f \)

■ Our program will execute in

\[
N \times \text{CPI} \times (1/f) = N \times \text{CPI} \times T \text{ seconds}
\]
How can I Make the Program Run Faster?

\[ N \times CPI \times (1/f) \]
How can I Make the Program Run Faster?

\[ N \times CPI \times (1/f) \]

- Reduce the number of instructions
  - Make instructions that ‘do’ more (CISC)
  - Use better compilers
How can I Make the Program Run Faster?

\[ N \times CPI \times (1/f) \]

- **Reduce the number of instructions**
  - Make instructions that ‘do’ more (CISC)
  - Use better compilers

- **Use less cycles to perform the instruction**
  - Simpler instructions (RISC)
  - Use multiple units/ALUs/cores in parallel
How can I Make the Program Run Faster?

\[ N \times CPI \times (1/f) \]

- Reduce the number of instructions
  - Make instructions that ‘do’ more (CISC)
  - Use better compilers

- Use less cycles to perform the instruction
  - Simpler instructions (RISC)
  - Use multiple units/ALUs/cores in parallel

- Increase the clock frequency
  - Find a ‘newer’ technology to manufacture
  - Redesign time critical components
  - Adopt pipelining
Single-Cycle Performance

- $T_C$ is limited by the critical path (1w)
Single-Cycle Performance

- Single-cycle critical path:
  - $T_c = t_{pcq_{PC}} + t_{mem} + \max(t_{RFread}, t_{sext} + t_{mux}) + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$

- In most implementations, limiting paths are:
  - memory, ALU, register file.
  - $T_c = t_{pcq_{PC}} + 2t_{mem} + t_{RFread} + t_{mux} + t_{ALU} + t_{RFsetup}$
Single-Cycle Performance Example

<table>
<thead>
<tr>
<th>Element</th>
<th>Parameter</th>
<th>Delay (ps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register clock-to-Q</td>
<td>$t_{pcq_PC}$</td>
<td>30</td>
</tr>
<tr>
<td>Register setup</td>
<td>$t_{setup}$</td>
<td>20</td>
</tr>
<tr>
<td>Multiplexer</td>
<td>$t_{mux}$</td>
<td>25</td>
</tr>
<tr>
<td>ALU</td>
<td>$t_{ALU}$</td>
<td>200</td>
</tr>
<tr>
<td>Memory read</td>
<td>$t_{mem}$</td>
<td>250</td>
</tr>
<tr>
<td>Register file read</td>
<td>$t_{RFread}$</td>
<td>150</td>
</tr>
<tr>
<td>Register file setup</td>
<td>$t_{RFsetup}$</td>
<td>20</td>
</tr>
</tbody>
</table>

$$T_c =$$
## Single-Cycle Performance Example

<table>
<thead>
<tr>
<th>Element</th>
<th>Parameter</th>
<th>Delay (ps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register clock-to-Q</td>
<td>$t_{pcq_PC}$</td>
<td>30</td>
</tr>
<tr>
<td>Register setup</td>
<td>$t_{\text{setup}}$</td>
<td>20</td>
</tr>
<tr>
<td>Multiplexer</td>
<td>$t_{\text{mux}}$</td>
<td>25</td>
</tr>
<tr>
<td>ALU</td>
<td>$t_{\text{ALU}}$</td>
<td>200</td>
</tr>
<tr>
<td>Memory read</td>
<td>$t_{\text{mem}}$</td>
<td>250</td>
</tr>
<tr>
<td>Register file read</td>
<td>$t_{\text{RFread}}$</td>
<td>150</td>
</tr>
<tr>
<td>Register file setup</td>
<td>$t_{\text{RFsetup}}$</td>
<td>20</td>
</tr>
</tbody>
</table>

\[
T_c = t_{pcq\_PC} + 2t_{\text{mem}} + t_{\text{RFread}} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{RFsetup}}
\]

\[
= [30 + 2(250) + 150 + 25 + 200 + 20] \text{ ps}
\]

\[
= 925 \text{ ps}
\]
Single-Cycle Performance Example

Example:

For a program with 100 billion instructions executing on a single-cycle MIPS processor:
Single-Cycle Performance Example

Example:

For a program with 100 billion instructions executing on a single-cycle MIPS processor:

\[
\text{Execution Time} = \# \text{ instructions} \times \text{CPI} \times \text{TC}
\]

\[
= (100 \times 10^9)(1)(925 \times 10^{-12} \text{ s})
\]

\[
= 92.5 \text{ seconds}
\]